



# TheMida : defeating ring0

deroko/ARTeam

Version 1.0 – May 2006

1.	Abstract.....	2
2.	Know your enemy and dump it.....	3
3.	Imports, IAT and I.....	14
4.	Foo Bar Baz code .....	19
5.	Conclusion .....	30
6.	References/Tools.....	31
7.	Greetings.....	32

## Keywords

Themida, unpacking



## 1. Abstract

TheMida 1.0.0.5 with offensive ring0 driver is not very used nowadays. But still it doesn't mean that we shouldn't pay attention to it. Even more, when something is not researched and written about it is more fun to play with such protection. I have nothing more to say about it, I will show you my research and all I have done to repair dump.

My target is Advanced Submitter 4.1.4 [3]

Of course, to play with themida 1.0.0.5 we are going to use some tools:

- IDA
- LiveKd
- wARK
- hiew
- tasm32

Forget about debugger it is not useful while playing with themida 1.0.0.5 and ring0 hooks.

**S verom u Boga, deroko/ARTeam**



## 2. Know your enemy and dump it

Protection implemented by oreans.sys is weird it hooks some SSDT entries, IDT and some exports of ntoskrnl.exe:

```
SSDT entry 0035 hooked by : 0xF3B7ACBE - NtCreateThread
SSDT entry 003A hooked by : 0xF3B7B1A0 - NtDebugContinue
SSDT entry 00B2 hooked by : 0xF3B7AACA - NtQueryVirtualMemory
SSDT entry 00BA hooked by : 0xF3B7A014 - NtReadVirtualMemory
SSDT entry 0101 hooked by : 0xF3B7A9D0 - NtTerminateProcess
SSDT entry 0115 hooked by : 0xF3B7A000 - NtWriteVirtualMemory
```

Now we will check hooks in exports of ntoskrnl.exe:

```
C:\>scanhook.exe
scanhook      - (c) 2006 deroko/ARTeam
Hooked : KeAttachProcess
Hooked : vsprintf
C:\>
```

Well KeAttachProcess is hooked because themida want to deny any pte changing from ring0 and process dumping from ring0. Well it doesn't hook KeStackAttachProcess so we can still attach to process from ring0 and dump it, but we are also able to emulate KeAttachProcess very simple.

Now let's see what we have in IDT:

01h	0008:FFFFFFFF	Interrupt	32 bit	03	0
03h	0008:FFFFFFFF	Interrupt	32 bit	03	0
0Bh	0008:EBF0E3E8	Interrupt	32 bit	00	1
0Eh	0008:EBF0E000	Interrupt	32 bit	00	1

Oki, int1 and int3 are "hooked" with 0xFFFFFFFF, the reason why themida hooks int 0eh is to catch 0xFFFFFFFF memory access and according to instruction that caused access to 0xFFFFFFFF to transfer execution to KiTrap01 or KiTrap03, in other words, default handlers. Of course if it finds int3h in themida protected process, my best guess is BSOD because I didn't want to experiment and to cause another BSOD just to learn in hard way to do not mess with themida hooks J

If you try to remove any of these hooks you will get BSOD.

So to understand themida hooks I used one small home made ring0 memory dumper to dump hooks. But before we move to hook dumps I will show you hooks for KeAttachProcess and vsprintf in ntoskrnl.exe dumped with LiveKd:

```
kd> u KeAttachProcess
nt!KeAttachProcess:
804e3173 e9884e5d78      jmp     f8ab8000
804e3178 56                  push    esi
804e3179 57                  push    edi

kd> u ntoskrnl!vsprintf
nt!vsprintf:
8050716b 33c0                xor     eax,eax
8050716d c3                  ret
```



Oki, if we dump memory content of hook stored in KeAttachProcess we will see this code:

```
push    ebp
mov     ebp, esp
pusha
call    $+5
pop     edx
sub     edx, 940FB78h
push    fs                ; save FS
mov     eax, 30h          ; make it point to kpcr
mov     fs, ax            ;
mov     eax, large fs:124h ; grab CurrentThread from kpcr
mov     eax, [eax+44h]     ; grab EPROCESS from ETHREAD
mov     [edx+940FC00h],    eax ; save EPROCESS struct
pop     fs                ; restore fs
mov     eax, [ebp+4]       ; now it takes saved EIP from stack
cmp     eax, [edx+940FBF0h] ; this part is not interesting for us
jbe     short loc_F8AB8040 ; because it is junk
cmp     eax, [edx+940FBF4h] ;
jnb     short loc_F8AB8040 ;
jmp     short loc_F8AB807D ;
; ~~~~~

loc_F8AB8040:
cmp     eax, [edx+940FBF8h] ;still junk
jbe     short loc_F8AB8052
cmp     eax, [edx+940FBFCh]
jnb     short loc_F8AB8052
jmp     short loc_F8AB807D
; ~~~~~

loc_F8AB8052:
mov     eax, [ebp+8]        ;oki now we get to good stuff
cmp     eax, [edx+940FC00h] ;cmp passed EPROCESS to KeAttachProcess
jz      short loc_F8AB8076 ;with EPROCESS of protected process
mov     esi, 0F8AAC000h     ;themida internal struct
add     esi, 4              ;esi points to EPROCESS field in
                           ;internal struct

loc_F8AB8065:
cmp     dword ptr [esi], 47616420h ;is it signature?
jz      short loc_F8AB807D        ;if so end of loop
cmp     [esi], eax                ;EPROCESS of protected process
jz      short loc_F8AB8076        ;compared to saved one
add     esi, 4                    ;go to next entry in struct
jmp     short loc_F8AB8065        ;loop
; ~~~~~
; dumped structure comes here so it is easier to follow code
dd 47616420h                      ;signature (' daG')
dd 81B58DA0h                      ;EPROCESS of protected process
dd 47616420h                      ;signature again
dd 0
; ~~~~~

loc_F8AB8076:
;return w/o attaching to target process
popa
```



```
pop    ebp
xor     eax, eax
retn    4
```

```
loc_F8AB807D:                                ;call some code in oreans.sys
popa
pop     ebp
jmp     short loc_F8AB8095
```

```
loc_F8AB8095:
mov     edi, edi
push    ebp
mov     ebp, esp
jmp     near ptr 804E3178h    ;KeAttachProcess+5
```

To write driver that will be able to use KeAttachProcess we have to write small procedure that will be wrapper for KeAttachProcess, but this is completely unnecessary work because we may attach using KeStackAttachProcess but what the hell, we are doing this for fun and to learn something, at least that's the only reason why I'm playing with themida.

```
KeAttachProcessWrapper:
mov     edi, edi
push    ebp
mov     ebp, esp
iMOV    eax, KeAttachProcess
add     eax, 5
jmp     eax    ;execute KeAttachProcess+5
```

Simple isn't it J

This is very important instruction in above code:

```
mov     esi, 0F8AAC000h    ;themida internal struct
```

If we take a look in disassembly of hook in IDA we might see that this instruction is located at hook\_base + 5Dh:

```
seg000:0000005B      jz         short loc_76
seg000:0000005D      mov        esi, 0F8AAC000h
seg000:00000062      add        esi, 4
```

This structure is very important because it describes state of protected process, it has simple structure:

```
dd 47616420h    ;signature (' daG'
dd 81B58DA0h    ;EPROCESS of protected process
dd 47616420h    ;signature again
dd 0
dd 0
```



If we examine address 81B58DA0h in livekd we may notice that it is actually EPROCESS of protected application:

```
kd> dt nt!_EPROCESS 81b58da0
...
+0x168 PageDirectoryPte : _HARDWARE_PTE
+0x168 Filler           : 0
+0x170 Session          : 0xf8a55000
+0x174 ImageFileName    : [16] "AdvanceSubmitte" <--- application name
+0x184 JobLinks          : _LIST_ENTRY [ 0x0 - 0x0 ]
...
kd>
```

Voila we have researched internal themida struct.

Next important thing is that themida will remove hooks once protected application is done with its executing. To accomplish this themida uses hook of NtTerminateProcess so we are going to dump that memory region and analyze NtTerminateProcess hook:

```
push ebp
mov  ebp, esp
pusha
call $+5
pop  edx
sub  edx, 94552AFh
push edx
push 0
lea  eax, [edx+945531Fh]
push eax
push 0
mov  eax, 80559CD8h ;PsProcessType
xor  eax, eax ;eax set to 0
push eax
push 10h
push dword ptr [ebp+8]
mov  eax, 8055D468h
call eax ;ObReferenceObjectByHandle
pop  edx
cmp  dword ptr [edx+945531Fh], 0
jz   short loc_F364BA3A
mov  eax, [edx+945531Fh] ;EPROCESS of cur process
mov  ebx, eax
and  ebx, 7FFFFFFFh
mov  esi, 0F8AAC000h ; internal struct
```

```
loc_F364BA1D:
add  esi, 4
cmp  dword ptr [esi], 47616420h
jz   short loc_F364BA3A
cmp  [esi], eax
jz   short loc_F364BA32
cmp  [esi], ebx
jz   short loc_F364BA32
jmp  short loc_F364BA1D
```

```
; ~~~~~~
```



```
loc_F364BA32:
    mov     dword ptr [esi], 0FFFFFFFFh    ;signal to unhook probably
    jmp     short loc_F364BA42             ;what else could it be
; ~~~~~

loc_F364BA3A:
    popa
    pop     ebp
    push    80583E1Eh                      ;NtTerminateProcess
    retn
; ~~~~~

loc_F364BA42:
                                ;only return from
                                ;NtTerminateProcess
    popa
    pop     ebp
    xor     eax, eax
    retn     8
; ~~~~~
```

As you may see NtTerminateProcess will only set EPROCESS in internal themida struct to 0FFFFFFFFh which is signal probably for ring0 thread to signal some ring3 thread to call ExitProcess.

Oki, once themida process was loaded I was ready to play with themida internal struct and to set it to 0FFFFFFFFh as is done by themida itself when NtTerminateProcess is called.

I wrote driver for this particular target and used hook in KeAttachProcess to reach internal themida struct [keattachgame driver] note that this driver also hooks some entries in IDT:

```

    iMOV     esi, KeAttachProcess
    cmp     byte ptr[esi], 0e9h
    jne     __sh_fail

    mov     ecx, [esi+1]
    add     esi, 5
    add     esi, ecx

    push    esi
    pushs   <"Jump KeAttach 0x%.08X", 13, 10>
    iWin32  DbgPrint
    add     esp, 8

    add     esi, 5dh
    cmp     byte ptr[esi], 0beh
    jne     __sh_fail

    mov     esi, [esi+1]
    push    esi
    pushs   <"Internal struct 0x%.08X", 13, 10>
    iWin32  DbgPrint
    add     esp, 8

;
;terminate themdia protected process
;

    mov     [esi+4], 0ffffffffh
```



Loading and running this part of code in ring0 resulted in themida process termination. Next step for me was to prevent themida process termination by hooking NtTerminateProcess before themida process is loaded, because any modification of SSDT while themida is running will result in BSOD.

So I recoded my driver to load and wait ioctl codes sent to it by my programs to update PID of traced process:

```
myntterminateprocess    label    dword
                        push     ebp
                        mov      ebp, esp
                        sub      esp, 4
                        pusha

                        push     0
                        lea      eax, [ebp-4]
                        push     eax
                        push     1
                        iMOV     ebx, PsProcessType
                        push     dword ptr[ebx]
                        push     0
                        push     dword ptr[ebp+8]
                        iWin32   ObReferenceObjectByHandle
                        test     eax, eax
                        jnz      __failobjget

                        mov      eax, [ebp-4]
                        push     eax
                        iWin32   ObDereferenceObject

                        mov      eax, [ebp-4]
                        mov      eax, [eax.ep_UniqueProcessId]
                        cmp      eax, traced_pid
                        jne      __failobjget

                        push     eax
                        pushs     <"themdia process is terminating : 0x%.08X">
                        iWin32   DbgPrint
                        add      esp, 8

                        pushs     <"faking NtTerminateProcess for themida driver">
                        iWin32   DbgPrint
                        add      esp, 4

                        popa
                        mov      esp, ebp
                        pop      ebp
                        retn     8

__failobjget:          popa
                        mov      esp, ebp
                        pop      ebp
                        jmp      cs:[ntterminateprocess]
```

Basically I return from NtTerminateProcess if themida process is being terminated. Good, we run driver and wait. After we execute pid\_giver.asm which will notify NtTerminateProcess hook about PID and will tell driver to “play” with internal struct. Sooner or later we get exception at 1xxxxx trying to execute int 3h. Well, process is deprotected so we can dump it atm and load it in IDA. But





still I was wondering what was located at 1xxxxx when int 3h occurred so I dumped that region with LordPE and was more than shocked, it was virtually mapped kernel32.dll and suddenly occurrence of int3h became more than clear.

#### Disassembly of ExitProcess:

```
.text:7C81CAA2      mov     edi, edi
.text:7C81CAA4      push    ebp
.text:7C81CAA5      mov     ebp, esp
.text:7C81CAA7      push    0FFFFFFFFh
.text:7C81CAA9      push    77E8F3B0h
.text:7C81CAAE      push    dword ptr [ebp+8]
.text:7C81CAB1      call    loc_7C81C9FC
.text:7C81CAB6      jmp     near ptr unk_7C8399E4
```

First we have call to NtTerminateProcess and other stuff needed for nice and clean shutdown of a process. In normal case, our process will be terminated in call at 7C81C9FC and jmp will be never reached, but in this case, when I hook NtTerminateProcess to return without killing process, jmp at 7C81CAB6h is executed and we arrive here:

```
dword_7C8399E4  dd  0CCCCCCCCh, 8428E9CCh, 9090FFFCh
```

Bingo, there is our int 3h, so next step was to modify virtual kernel32.dll to call ExitThread once thread in ring3 tries to execute it. But to do this we have to hook CreateFileA. Before themida starts executing virtual kernel32.dll these files/devices are accessed:

```
\\.\SICE
\\.\SIWVID
\\.\NTICE
\\.\Global\XPROTECTOR
\\.\Global\XPROTECTOR
C:\WINDOWS\system32\KERNEL32.dll
C:\WINDOWS\system32\USER32.dll
C:\WINDOWS\system32\ADVAPI32.dll
\\.\Global\XPROTECTOR
C:\WINDOWS\system32\ntdll.dll
\\.\Global\XPROTECTOR
```

My next approach was to hook CreateFileA so it will load my modified kernel32.dll with this little modification:

```
.7C81CAA2| E90202FFFF      jmp     ExitThread  ---? (1)
.7C81CAA7: 6AFF      push    -1
.7C81CAA9: 68B0F3E877  push    077E8F3B0 ; 'wF=| '
```

Good, now I was ready to test my theory and to see if process will remain in memory when hooks are removed. Luckily my plan worked like a charm and I had deprotected process running in memory ready to be dumped without a problem. Of course, ring3 memory manager [5] is present and running all this time so I'm able to dump all allocated buffers in one file.

Then, of course I loaded file in Ida and after 30mins of analyzing and applying signatures I figured that I'm dealing with Delphi app.



When we are dealing with Delphi App we search for InitExe signature and I found it here:

```
___:00407408 ; __fastcall Sysinit::__linkproc__ InitExe(void *)
___:00407408 @Sysinit@@InitExe$qqrpv proc near
___:00407408         push     ebx
___:00407409         mov     ebx, eax
___:0040740B         xor     eax, eax
___:0040740D         mov     ds:TlsIndex, eax
___:00407412         push     0
___:00407414         call    sub_407344 ;call to GetModuleHandleA
___:00407419         mov     ds:dword_866718, eax
___:0040741E         mov     eax, ds:dword_866718
___:00407423         mov     ds:dword_85C094, eax
___:00407428         xor     eax, eax
___:0040742A         mov     ds:dword_85C098, eax
___:0040742F         xor     eax, eax
___:00407431         mov     ds:dword_85C09C, eax
___:00407436         call    @SysInit@_16395 ; SysInit::_16395
___:0040743B         mov     edx, (offset a123456789abcde+0Fh)
___:00407440         mov     eax, ebx
___:00407442         call    sub_4046C4
___:00407447         pop     ebx
___:00407448         retn
___:00407448 @Sysinit@@InitExe$qqrpv endp ; sp = -4
```

Now I knew what to hook to reach oep and to put bestard in infinite loops so I could inform driver to change themida internal struct, unload hooks, unload driver and attach to process with olly to get magic Delphi value and to find oep:

First I had to hook GetModuleHandleA to jmp \$ only if and only if it is called from certain locations:

```
.text:7C80B529         mov     edi, edi
.text:7C80B52B         push    ebp
.text:7C80B52C         mov     ebp, esp
.text:7C80B52E         cmp     [ebp+arg_4], 0
.text:7C80B532         jz      short loc_7C80B54C
.text:7C80B534         push    [ebp+arg_4]
.text:7C80B537         call    sub_7C80E2A4
.text:7C80B53C         test    eax, eax
.text:7C80B53E         jz      short loc_7C80B548
.text:7C80B540         push    dword ptr [eax+4]
.text:7C80B543         call    GetModuleHandleW
.text:7C80B548 loc_7C80B548:
.text:7C80B548         pop     ebp
.text:7C80B549         retn    4
.text:7C80B54C loc_7C80B54C:
.text:7C80B54C         mov     eax, large fs:18h
.text:7C80B552         mov     eax, [eax+30h]
.text:7C80B555         mov     eax, [eax+8]
.text:7C80B558         jmp     short loc_7C80B548
.text:7C80B558 GetModuleHandleA endp
```

I decided to hook after retn 4 because GetModuleHandleA is called from Delphi app with argument equal to 0. Oki I installed my hook:



```
__orig:    cmp     [ebp+4], 407419h
           jne     __orig
           jmp     $
           mov     eax, 400000h
           pop     ebp
           retn    4
```

Now when my hook is reached I signal driver to fake process termination, faked kernel32.dll is loaded and voila our process is running deprotected in memory, of course, use wARK to unload driver and attach to process with olly. You should check value of EBX because that will be your magic value and after a few step over in olly you reach your oep:

```
__ :0085BB01    call    @Sysinit@@InitExe$qqrpv
__ :0085BB06    mov     ebx, ds:off_86454C
__ :0085BB0C    mov     esi, ds:dword_864920
__ :0085BB12    mov     eax, [esi]
```

Bingo, now go a little bit up and assemble your little patch:

```
__ :0085BAFC    mov     eax, offset dword_85AF14
__ :0085BB01    call    @Sysinit@@InitExe$qqrpv
__ :0085BB06    mov     ebx, ds:dword_86454C
__ :0085BB0C    mov     esi, ds:dword_864920
__ :0085BB12    mov     eax, [esi]
```

Congratulation, you have dumped themida protected application at oep, now we have to fix imports and replaced code.

To be able to inject memory manager in virtual kernel32.dll I used int 80h hook to store jmp to my code in virtual \_k32.dll!VirtualAlloc, to load my kernel32.dll I hooked CreateFileA so it will give to themida .\myk32.dll instead of C:\windows\system32\kernel32.dll, all modifications are performed in my copy of kernel32.dll named myk32.dll:

```
.7C809A81| CD80          int     080
.7C809A83: 55          push    ebp
.7C809A84: 8BEC        mov     ebp, esp
.7C809A86: FF7514      push    d, [ebp][14]
.7C809A89: FF7510      push    d, [ebp][10]
.7C809A8C: FF750C      push    d, [ebp][0C]
.7C809A8F: FF7508      push    d, [ebp][08]
.7C809A92: 6AFF        push    -1
.7C809A94: E809000000 call    VirtualAllocEx ---? (1)
.7C809A99: 5D          pop     ebp
.7C809A9A: C21000      retn    00010 ;' ?'
```

Now I hooked int 80h in IDT and changed its DPL to 3 so instruction can be executed from ring3 without a problem:

```
initint
pushs     <"int 80h called", 13, 10>
iWin32    DbgPrint
add       esp, 4
```



```
mov     edi, [esp.int_eip]
sub     edi, 2

mov     al, 68h
stosb
mov     eax, memorymanager
stosd
mov     al, 0c3h
stosb

sub     [esp.int_eip], 2
restoreint
iretd
```

Now we know that call to VirtualAlloc in our virtual\_k32.dll will be redirected to memory manager. I also wrote one procedure to hook interrupts because I needed it during themida unpacking. Except int 80h I needed 2 more interrupts so writing same code 3 times would result in huge and unreadable source:

```
HookInterruptAdjustDpl proc
arg     intterupt_number:dword
arg     new_handler:dword
arg     old_handler:dword
pusha

push    edi
sidt    fword ptr[esp-2]
pop     edi

mov     eax, interrupt_number
shl     eax, 3

lea     edi, [edi+eax]
mov     ebx, old_handler

movzx   ecx, word ptr[edi+6]
rol     ecx, 16
mov     cx, word ptr[edi]

mov     [ebx], ecx

mov     ecx, new_handler

cli
mov     word ptr[edi], cx
rol     ecx, 16
mov     word ptr[edi+6], cx

movzx   ecx, word ptr[edi+4]
or      ch, 060h                ; dpl == 3
mov     word ptr[edi+4], cx
sti

popa
leave
retn    0ch
endp
```



And unhook procedure:

```
UnHookInterruptAdjustDpl proc
    arg    interrupt_number:dword
    arg    old_handler:dword
    pusha

    push    edi
    sidt    fword ptr[esp-2]
    pop     edi

    mov     eax, interrupt_number
    shl     eax, 3

    lea     edi, [edi+eax]

    mov     ecx, old_handler

    cli
    mov     word ptr[edi], cx
    rol     ecx, 16
    mov     word ptr[edi+6], cx
    movzx   ecx, word ptr[edi+4]
    and     ch, 9fh
    mov     word ptr[edi+4], cx    ;dpl == 0
    sti

    popa
    leave
    retn    8
endp
```

Usage:

```
push    offset oldint80handle
push    offset __newint80handle
push    80h
call    HookInterruptAdjustDpl
```

and:

```
push    oldint80handle
push    80h
call    UnHookInterruptAdjustDpl
```

That's all for this chapter, you might wanna check loader22.asm which has all stuff I used to dump themida and fix imports. Code is not very well optimized because I added code to it as I was advancing further in themida unpacking.



### 3. Imports, IAT and I

Import protection is very nice in themida, when I say nice, that means that it took me a lot of time to fix IAT. I will sum how import protection is performed:

First it will disassemble API and take 1<sup>st</sup> instruction and store it in separate buffer, then it will store garbage code and take next instruction and store it, and again garbage code, original byte, garbage code etc. if jmp/jcc is found it will be redirected to original jmp spot in destination API and if we have some API w/o jcc/jmp nor call to some procedure whole API with garbage code between original instructions will be copied to buffer allocated by protector, of course, jcc/jmp are redirected and garbage is stored after them so next instruction that follows them is obfuscated till ret/retn.

Of course, jmp dword ptr [api] are changed with jmp \_buffer:

```
____:00407408      push     ebx
____:00407409      mov     ebx, eax
____:0040740B      xor     eax, eax
____:0040740D      mov     ds:TlsIndex, eax
____:00407412      push    0
____:00407414      call    sub_407344      <---- call GetModuleHandleA

...

____:00407344 sub_407344      proc near
____:00407344      jmp     near ptr 2FC0E78h
____:00407344 sub_407344      endp
```

As you may see there is relative jmp (e9) instead of jmp dword ptr [] (ff25h). Of course, fixing relative jmps is not problem, and to automate process of their fixing I wrote iatfix.asm which is supplied with this document.

Bigger problem is how to prevent themida from obfuscating APIs, after a 5-6 bsods I finally got good solution.

Let me tell you something, with memory manager injected I got all buffers in memory including obfuscated APIs, to get partially working dump I injected dll loader in last section which will load used dlls by target and in such way make all jmp/jcc in obfuscated apis valid. Now we may run application and it should work. For this code please refer to [addsec.asm] supplied with this document.

Now after I run [iatfix.asm] I got everything as I wanted it to be:

```
.dumped:00407344 sub_407344      proc near
.dumped:00407344      jmp     ds:off_1BD50E4
.dumped:00407344 sub_407344      endp
.dumped:00407344
.dumped:0040734A      mov     eax, eax
.dumped:0040734C
.dumped:0040734C sub_40734C      proc near
.dumped:0040734C      jmp     ds:off_1BD50E8
.dumped:0040734C sub_40734C      endp
```



For all olly lovers, jmp ds:off\_ is actually jmp dword ptr[] aka ff25, now I had everything I needed. Working code that will fix jmps but still no valid API pointers. Now I can run importrec and find modified IAT but still no valid APIs.

Since themida obfuscates API by stealing instruction and inserting garbage in its own buffer I planned to hook all exports of loaded dlls with code that will help me later on with importrec plug-in to find APIs. My API hook is very simple. [Check hook4.asm sample code]

From this:

```
.text:7C80B529      mov     edi, edi    <-----+
.text:7C80B52B      push    ebp         |
.text:7C80B52C      mov     ebp, esp    <-----+
.text:7C80B52E      cmp     [ebp+arg_4], 0
.text:7C80B532      jz      short loc_7C80B54C
.text:7C80B534      push    [ebp+arg_4]
.text:7C80B537      call    sub_7C80E2A4
```

Api becomes:

```
.text:7C80B529      jmp     my_hook
.text:7C80B52E      cmp     [ebp+arg_4], 0
.text:7C80B532      jz      short loc_7C80B54C
.text:7C80B534      push    [ebp+arg_4]
.text:7C80B537      call    sub_7C80E2A4
```

And my hook is :

```
00F297E2      50                PUSH EAX
00F297E3      B8 29B5807C       MOV EAX, kernel32.GetModuleHandleA
00F297E8      58                POP EAX
00F297E9      8BFF             MOV EDI, EDI
00F297EB      55                PUSH EBP
00F297EC      8BEC             MOV EBP, ESP
00F297EE      - E9 3B1D8E7B     JMP kernel32.7C80B52E
```

So later on my importrec plug-in should only trace till mov eax, API and return valid API pointer J  
Luckily TheMida didn't try to obfuscate API when jmp at its entry is found so after running my hook engine I got valid pointers:

```
00407412      6A 00             PUSH 0
00407414      E8 2BFFFFFF       CALL AdvanceS.00407344
```

We trace this call:

```
00407344      - E9 E041407C     JMP kernel32.GetModuleHandleA
```

Ohhh, clap, clap J

```
7C80B529 >- E9 B4E27184     JMP AdvanceS.00F297E2
7C80B52E      837D 08 00       CMP DWORD PTR SS:[EBP+8], 0
```



And jmp goes to hook and back to code.

To avoid hooking all dlls, the ones loaded by target and those loaded by other dlls we have to hook LoadLibraryA and GetModuleHandleA but we are also aware that there is virtual kernel32.dll. So to hook those two I installed two more interrupts with DPL 3 [keattachgame driver]:

GetModuleHandleA:

```
.7C80B529| CD79          int      079
.7C80B52B: 55          push    ebp
.7C80B52C: 8BEC        mov     ebp,esp
.7C80B52E: 837D0800    cmp     d,[ebp][08],0
```

and LoadLibraryA:

```
.7C801D77| CD81          int      081
.7C801D79: 55          push    ebp
.7C801D7A: 8BEC        mov     ebp,esp
.7C801D7C: 837D0800    cmp     d,[ebp][08],0
```

My hooks in ring0 are similar to above one used for memory manager so now we are sure that every dll that is being loaded will be hooked J

Also I avoid hooking of ntdll.dll because if I store jmp at entry of Native APIs themida will BSOD my system. BSOD comes from simple fact that themida expects to find valid service number in a few exported APIs from ntdll.dll (the ones hooked in SDT) if we store “jmp hook” reading service number will result in wrong number, and writing to wrong address in ring0 is equal to bye, bye my dear system.

Now driver is loaded, and we are ready to watch output of DebugView[2]:

```
00000000    0.00000000  Oreans driver loaded in memory (v1.40)
00000001    0.00000112
00000002    0.14517654  int 80h called
00000003    0.42267776  int 79h called
00000004    0.44069371  [2080]
00000005    0.44069371  [2080]
00000006    0.44069371  [2080]
00000007    0.44069371  [2080] -----
00000008    0.44069371  [2080] ---          Themida Professional          ---
00000009    0.44069371  [2080] ---          (c)2005 Oreans Technologies          ---
00000010    0.44069371  [2080] -----
00000011    0.44069371  [2080]
00000012    0.44069371  [2080]
00000013    0.46072280  int 81h called
00000014    0.46082562  [2080] loadlibrarya hooked?
00000015    0.46086192  [2080] dll loaded 0x7C900000
00000016    0.48601010  [2080] loadlibrarya hooked?
00000017    0.51330322  [2080] dll loaded 0x7C900000
00000018    0.52680326  [2080] loadlibrarya hooked?
00000019    0.52794141  [2080] dll loaded 0x7C900000
00000020    0.53012294  [2080] hooking damn dll
00000021    0.53035456  [2080] dll hooked : 0x7C800000
00000022    0.54317516  [2080] hooking damn dll
00000023    0.54372722  [2080] dll hooked : 0x77D40000
00000024    0.55571115  [2080] hooking damn dll
```





```
00000025    0.55580807 [2080] dll hooked : 0x77DD0000
00000026    0.57459623 [2080] loadlibrarya hooked?
00000027    0.57480603 [2080] dll loaded 0x77120000
00000028    0.57484204 [2080] hooking damn dll
00000029    0.57495522 [2080] dll hooked : 0x77120000
00000030    0.59113181 [2080] loadlibrarya hooked?
00000031    0.59117877 [2080] dll loaded 0x71B20000
00000032    0.59121144 [2080] hooking damn dll
00000033    0.59145534 [2080] dll hooked : 0x71B20000
00000034    0.59156036 [2080] loadlibrarya hooked?
00000035    0.59159249 [2080] dll loaded 0x77C00000
00000036    0.59162354 [2080] hooking damn dll
00000037    0.59174669 [2080] dll hooked : 0x77C00000
00000038    0.59180927 [2080] hooking damn dll
00000039    0.59224790 [2080] dll hooked : 0x77F10000
00000040    0.60678184 [2080] hooking damn dll
00000041    0.60691762 [2080] dll hooked : 0x774E0000
00000042    0.62314314 [2080] hooking damn dll
00000043    0.62323815 [2080] dll hooked : 0x5D090000
00000044    0.63646275 [2080] loadlibrarya hooked?
00000045    0.63650769 [2080] dll loaded 0x76390000
00000046    0.63654011 [2080] hooking damn dll
00000047    0.63662922 [2080] dll hooked : 0x76390000
00000048    0.64198887 [2080] loadlibrarya hooked?
00000049    0.64205229 [2080] dll loaded 0x73000000
00000050    0.65171999 [2080] hooking damn dll
00000051    0.65182054 [2080] dll hooked : 0x73000000
00000052    0.66523176 [2080] loadlibrarya hooked?
00000053    0.66548234 [2080] dll loaded 0x7C9C0000
00000054    0.66551697 [2080] hooking damn dll
00000055    0.66563183 [2080] dll hooked : 0x7C9C0000
00000056    0.69346023 [2080] loadlibrarya hooked?
00000057    0.69481510 [2080] dll loaded 0x771B0000
00000058    0.69487798 [2080] hooking damn dll
00000059    0.69498020 [2080] dll hooked : 0x771B0000
00000060    0.72827160 [2080] loadlibrarya hooked?
00000061    0.72832191 [2080] dll loaded 0x77260000
00000062    0.72835487 [2080] hooking damn dll
00000063    0.72844315 [2080] dll hooked : 0x77260000
00000064    0.73640925 [2080] loadlibrarya hooked?
00000065    0.73645365 [2080] dll loaded 0x763B0000
00000066    0.73648578 [2080] hooking damn dll
00000067    0.73663414 [2080] dll hooked : 0x763B0000
00000068    1.11949956 [2080] loadlibrarya hooked?
00000069    1.11961353 [2080] dll loaded 0x7C900000
00000070    1.12467396 [2080] loadlibrarya hooked?
00000071    1.13394499 [2080] dll loaded 0x7C900000
00000072    1.19805110 [2080] loadlibrarya hooked?
00000073    1.19807744 [2080] dll loaded 0x7C800000
```

Now when last hook is reached we may check in Process Explorer from [www.sysinternals.com](http://www.sysinternals.com) or in Task Manager that our process is killing our CPU due to hook in GetMoudleHandleA (jmp \$) and now we will inform driver to simulate process exit and we watch output in DebugView [pid\_giver.asm]:



```
00000072 42.45299149 Jmp KeAttach 0xF8AC8000
00000073 42.45300674 Internal struct 0xF8AC2000
00000074 45.15629578 themdia process is terminating : 0x000009B4
00000075 45.15634918 faking NtTerminateProcess for themida driver
```

Wait a few seconds and use wARK to unload driver (usually I check IDT using wARK to know if hooks are removed, very useful toolJ ):

```
00000076 227.15957642 Oreans driver unloaded
```

Now dump process and run addsec and iatfix and voila, there is only couple of wrong APIs which are not hooked:

- forwarded ones to ntdll.dll
- ntdll.dll APIs
- the ones that have call/jmp/jcc in 1<sup>st</sup> 5 bytes of API (some exports from user32.dll)

Of course, I will recode my loader to hook everything except ntdll.dll and to handle forwarded APIs from kernel32.dll in an appropriate way:

```
01B050C8 7C809481 kernel32.VirtualAlloc
01B050CC 7C809B14 kernel32.VirtualFree
01B050D0 7C809FA1 kernel32.InitializeCriticalSection
01B050D4 7C80901F ASCII "NTDLL.RtlEnterCriticalSection"
01B050D8 7C80912B ASCII "NTDLL.RtlLeaveCriticalSection"
01B050DC 7C808FCC ASCII "NTDLL.RtlDeleteCriticalSection"
01B050E0 77D6FA46 USER32.GetKeyboardType
01B050E4 7C80B529 kernel32.GetModuleHandleA
01B050E8 7C8099BD kernel32.LocalAlloc
01B050EC 7C809750 kernel32.TlsGetValue
01B050F0 7C809BF5 kernel32.TlsSetValue
01B050F4 77DD6BF0 ADVAPI32.RegCloseKey
```

Now all I have to do is to save import tree.txt, modify it a little bit and inject same apiloader as one presented in unpacking armadilloed dll [1] and after we run injected loader we get these pointers:

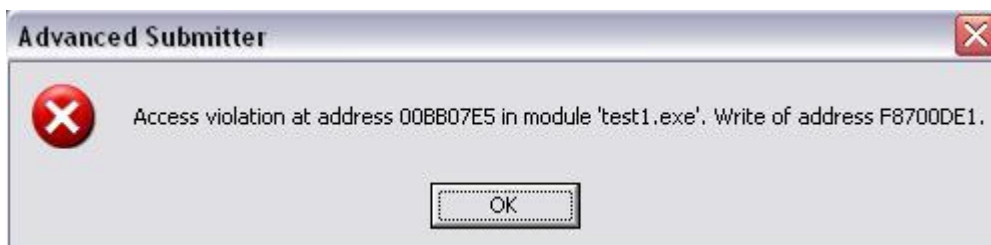
```
01B050CC 7C809B14 kernel32.VirtualFree
01B050D0 7C809FA1 kernel32.InitializeCriticalSection
01B050D4 7C901005 ntdll.RtlEnterCriticalSection
01B050D8 7C9010ED ntdll.RtlLeaveCriticalSection
01B050DC 7C91188A ntdll.RtlDeleteCriticalSection
01B050E0 77D6FA46 user32.GetKeyboardType
01B050E4 7C80B529 kernel32.GetModuleHandleA
01B050E8 7C8099BD kernel32.LocalAlloc
01B050EC 7C809750 kernel32.TlsGetValue
01B050F0 7C809BF5 kernel32.TlsSetValue
01B050F4 77DD6BF0 ADVAPI32.RegCloseKey
```

That's all about import fixing. Check [apiloader.asm] for complete code and dumped iat.



## 4. Foo Bar Baz code

Run application and you will see beauty of Delphi application, it will inform you about address where exception is caused, in my case it was this message:



It seems that themida is trying to read memory from F8700DE1 which is obviously ring0 memory and ring3 application w/o gateway to ring0 can't access it. If you keep following this address you will see that process is somehow accessing to r0 memory because you may also see how handlers for int1 and int3 are installed in your ring3 code, once handler is installed you will see int1 and int3 in code which should transfer execution to handlers in memory < 80000000h. Weird, but if we check this address in IDA we may see a few interesting stuff:

```
.dumped:00BB07CD      pusha
.dumped:00BB07CE      call     $+5
.dumped:00BB07D3      pop      ebp
.dumped:00BB07D4      sub      ebp, 945B90Eh
.dumped:00BB07DA      mov      esi, [ebp+9350759h]
.dumped:00BB07E0      mov      eax, 2
.dumped:00BB07E5      loc_BB07E5:
.dumped:00BB07E5      xchg     al, [esi]
```

There you go, xchg al, [esi] is causing exception. Now if we follow xrefs to this code in IDA we will endup here:

```
.dumped:00826C6D      call     @Db@TWideStringField@GetAsString$qgrv
.dumped:00826C72      call     near ptr sub_BB07CD
.dumped:00826C77      or      eax, [eax]
.dumped:00826C77 ; -----
.dumped:00826C79      dd      0
.dumped:00826C7D      dd      230000h
.dumped:00826C81      dd      15200000h
.dumped:00826C85      dd      1E153B40h
.dumped:00826C89      dd      988AC38Fh
.dumped:00826C8D      dd      5B58CDD3h
.dumped:00826C91      dd      68DF40EAh
.dumped:00826C95      dd      1F890FA3h
.dumped:00826C99      dd      453723EBh
.dumped:00826C9D      dd      2C7234A8h
.dumped:00826CA1      dd      0B8AFCB1h
.dumped:00826CA5      dd      1000000h
.dumped:00826CA9      dd      23000000h
.dumped:00826CAD      dd      20000000h
.dumped:00826CB1 ; -----
.dumped:00826CB1      mov      edx, ds:dword_87FFC4
.dumped:00826CB7      mov      eax, 826DB4h
.dumped:00826CBC      call     unknown_libname_68
```



If we step into procedure at BB07CDh we might see how it reads from ring0 memory and how int1 and int3 entries in IDT are changed from ring3 code to point to some code in ring3. At first I thought that I'm crazy and I don't see well but honestly there is code that sets IDT entries from ring3. I was wondering how this is performed. I dumped cs/ds registers just to make sure that those are ring3. Then I wrote small driver to walk through memory addresses from 80000000h – FFFFFFFFh and check PDE/PTE for User/Supervisor flag set in them. If 3<sup>rd</sup> bit of PDE or PTE, depending on page size, is set that page can be accessed from any ring that is lower than ring0 (lower DPL) so after running my driver I got this from DebugView:

```
00000000      0.00000000  Page at 0x8003F000 user visible
```

Huh, 0x8003F000 is base of IDT and is visible/writable/present in PTE so this address can be accessed by any ring3 program without problem. Themida will deny access to this memory to any ring3 process except protected one, of course.

Then to test my theory I recoded int0e handler used by themida to allow access to any ring3 application to play with IDT (write/read) and set int1/int3 handlers. Here is disassembly of themida int0e handler [check intfoobar driver]:

```
mov     eax, [esp+28h]    ;eip in eax
cmp     eax, 0FFFFFFFh   ;is it caused by eip = ffffffff in other words
jz      short loc_39      ;caused by execution of int1 and int3
mov     eax, [esp+24h]    ;takes ErrorCode
cmp     eax, 0FFFFFFFh   ;errorcode can't be -1 so this jmp is taken
jnz     loc_1DD
```

...

```
mov     eax, [ebx]        ;this takes EIP passed to int1/int3
cmp     byte ptr [eax-1], 0CCh ;and check if -1 is caused by int3h
jz      short loc_1E5      ;pass exception to _KiTrap03
cmp     word ptr [eax-2], 3CDh ;caused by INT 3h
jz      short loc_1E5      ;pass exception to _KiTrap03
cmp     word ptr [eax-2], 1CDh ;caused by INT 1h
jz      short loc_1B1      ;pass Exception to _KiTrap0D
popa
popf
add     esp, 10h
jmp     short loc_1F0      ;pass Exception to _KiTrap01
```

This code is very simple, since int1 and int3 handles are set to 0FFFFFFFh each occurrence of int3h (cc, cd03) from ring3 will try to call 0FFFFFFF which will result in calling page fault handler (int 0eh), also if Trap Flag is set in eflags or drX registers are used it will result in 0FFFFFFF access. Calling int 1 (cd01) will also result in 0FFFFFFF because themida sets int1 handler to DPL 3, and in normal case its DPL is 0 which means each occurrence of int1 in ring3 will result in calling Global Protection fault aka \_KiTrap0D or int 0d handler (remember SoftIce detection via int1 trick?). Oki, here is how it works:



## Int1/int3 – stack layout

```
+-----+
| Eflags |
+-----+
| CS      |
+-----+
| EIP     |
+-----+
```

Now it tries to access address at 0FFFFFFFFh and page fault is generated so stack looks like this:

```
+-----+
| Eflags | saved from int1/int3
+-----+
| CS      | saved from int1/int3
+-----+
| EIP     | saved from int1/int3
+-----+
| Eflags | saved for int 0e
+-----+
| CS      | saved for int 0e
+-----+
| EIP     | EIP == 0FFFFFFFFh
+-----+
| ErrorCode | ErrorCode
+-----+
```

Now if int 0e handler detects that access to 0FFFFFFFFh is caused by certain conditions as shown in above disassembly it will align stack, eg. remove data passed to int 0e handler and call KiTrap01/KiTrap03/KiTrap0D depending on instruction that caused access violation. For fun I wrote small driver to do the same thing as themida and you may find it in [intfoobar] folder witch will grant any ring3 application to read/write IDT and also will set int1/int3 handlers to 0FFFFFFFFh. Run your debugger and debug any application w/o problem. Difference between my driver and oreans.sys is that oreans.sys will BSOD your system if int1/int3 exception occurs in protected process. You may change protection in themida protected application and inject nonintrusive tracers in it to see what is happening with code once we reach code caves.

We MAY use PAGE\_GUARD in protected process to trace its execution. PAGE\_GUARD, PAGE\_NOACCESS and other page attributes are implemented in Windows memory manger trough Page File. If we trace VirtualProtect with any “weird” attributes set that are not CPU specific we may see how page is erased from PTE and P bit is set to 0, also ntoskrnl.exe generates some “internal protection mask”, stores PFN in PTE and uses invlpg to flush TLB, making sure that any access to PAGE goes trough int0e handler. Once \_KiTrap0E gains control it will check “internal protection mask” and decide what exception to throw back to ring3. If no exception is needed PAGE is brought back to memory and PTE is set to point to right Physical frame and P flag is set to 1. Playing with PFN saved in PTE is kinda dangerous, we don’t know if Microsoft © is going to change PFN format so we can’t write any generic tool to check for presence of PAGE\_GUARD, in such way themida int 0e handler doesn’t know if exception occurred due to PAGE\_NOACCESS or PAGE\_GUARD. For more on this subject please refer to [6].



**Role of int 0e handler hook is to prevent system from crashing if some other application access int1/int3, and to grant access to the themida protected process to ring0 memory allocated by oreans.sys and to IDT as well.**

Now when we know theory we should try that in practice, and see if everything worked as we planned. For this I have used again hook in GetModuleHandleA to reach OEP, but now, I redirect execution to my code which will prepare nonintrusive tracer.

Before we move to nonintrusive tracer for themida protected application, you must be aware that we are not allowed to use int1/int3 in it or we will get BSOD. So how to step over instruction which generated exception? Simple, use some other instruction, privileged instruction. I used CLI, its usage in ring3 will generate privileged instruction exception and its size is one byte (0fah), same size as int3h (0cch).

Oki lets take a look at our crypted code again:

```
.dumped:0082E03E      call     @System@@LStrAsg$qqrV
.dumped:0082E043      call     near ptr sub_BB07CD
.dumped:0082E048      or       eax, [eax]
.dumped:0082E048 ; -----
.dumped:0082E04A      db       0
.dumped:0082E04B      db       0
.dumped:0082E04C      db       0
.dumped:0082E04D      db       0
.dumped:0082E04E      db       0
.dumped:0082E04F      db       0
.dumped:0082E050      db       3Ah
.dumped:0082E051      db       1
.dumped:0082E052      db       0
.dumped:0082E053      db       0
.dumped:0082E054      db       20h
.dumped:0082E055      dd       8F20EF73h
.dumped:0082E059      dd       0AF22EC81h
.dumped:0082E05D      dd       8ACBEB98h
.dumped:0082E061      dd       8ACBEB98h

...
.dumped:0082E18D      db       0
.dumped:0082E18E      db       0
.dumped:0082E18F      db       0
.dumped:0082E190      db       1
.dumped:0082E191      db       0
.dumped:0082E192      db       0
.dumped:0082E193      db       0
.dumped:0082E194      db       3Ah
.dumped:0082E195      db       1
.dumped:0082E198      db       20h
.dumped:0082E199 ; -----
.dumped:0082E199      mov      cl, 1
.dumped:0082E19B      mov      edx, esi
.dumped:0082E19D      mov      eax, [ebp-4]
```

Call at 0082E043h will actually leave on stack address of crypted code and jmp to decrypt proc. Also you might see how there is same code marker at 0082E048 and 0082E18D, don't be confused with or eax, [eax] it is also signature but disassembled by IDA J



My next approach was to change `call near ptr sub_BB07CD` with `jmp` to my hook which will initialize nonintrusive tracer, store good ret address on stack and wait for access in code section. Actually I didn't know at first that this code is crypted, my original intension was to log which part of code are accessed, what register values re used at that point, and then, later on, to emulate those calls. After running tracer for 1<sup>st</sup> time I was more then happy, I had decrypted code. After I saw that my approach is working I created two loaders: `loader_bingo` and `loader_type3`. Loader bingo is supposed to tell me from where starts decrypted code, and `loader_type3` is used later on to dump that memory region. There is totally 5 code crypt blocks, 3 during initialization and 2 during gallery creation. [All of dumped blocks are supplied with this document]

82E055h  
82E6C3h  
855361h  
826C84h  
82740Ch

Let's take a look at important parts of nonintrusive tracer for this application:

```

                                cmp     [ebp+4], 407419h
                                je      __makefun
                                mov     eax, 400000h
                                pop     ebp
                                retn    4

__makefun:                     pusha
                                call     newcodedelta
newcodedelta:                  pop     ebp
                                sub     ebp, offset newcodedelta

                                mov     edi, 082e043h

                                lea     ebx, [ebp+test2]

                                mov     ecx, edi
                                add     ecx, [edi+1]
                                add     ecx, 5

                                mov     dword ptr[ebp+ring0ring3address+1], ecx

                                mov     ecx, edi
                                mov     byte ptr[edi], 0e9h
                                add     ecx, 5
                                sub     ebx, ecx
                                mov     dword ptr[edi+1], ebx

                                popa
                                mov     eax, 400000h
                                pop     ebp
                                retn    4
```

This part of code is called from `GetModuleHandleA` hook, and is used to check when we are close to our OEP. When we reach OEP we will hook 1<sup>st</sup> call to themida decrypt code and redirect execution to our code which will initialize nonintrusive tracer:



```
test2          label    dword
               pusha
               call     test2delta
test2delta:    pop      ebp
               sub      ebp, offset test2delta

               mov      [esp+8*4], 0082E048h

               pushs    <"ntdll.dll">
               call     [ebp+GetModuleHandleA]
               mov      [ebp+ntdll], eax

               mov      ebx, [eax+3ch]
               add      ebx, eax

               pushv    <dd      ?>
               push     PAGE_EXECUTE_READWRITE
               push     [ebx.pe_sizeofimage]
               push     eax
               call     [ebp+VirtualProtect]

               gethash  <NtContinue>
               push     hash
               push     [ebp+ntdll]
               call     getprocaddress
               mov      [ebp+NtContinue], eax

               gethash  <KiUserExceptionDispatcher>
               push     hash
               push     [ebp+ntdll]
               call     getprocaddress
               mov      [ebp+KiUserExceptionDispatcher], eax

               mov      edi, eax
               lea      ebx, [ebp+nonintrusive_logger]

               mov      ecx, edi
               mov      byte ptr[edi], 0e9h
               add      ecx, 5
               sub      ebx, ecx
               mov      dword ptr[edi+1], ebx

               add      edi, 7
               mov      dword ptr[ebp+retkiuser+1], edi

               pushv    <dd      ?>
               push     PAGE_EXECUTE_READWRITE or PAGE_GUARD
               push     45bb01h
               push     400000h
               call     [ebp+VirtualProtect]

               popa

ring0ring3address: push    0deadc0deh
               retn
```





test2 code is here used to initialize nonintrusive tracer, 1<sup>st</sup> we change ret address on stack to point to crypted code, then we retrieve base of ntdll.dll and change protection on it to PAGE\_EXECUTE\_READWRITE, then we get address of KiUserExceptionDispatcher and NtContinue because we need both for nonintrusive tracer. Now we insert hook in KiUser as I did in ExeCryptor and Anit-anti-dump and nonintrusive tracers articles [4, 5] and we wait for a little bit of magic.

Nonintrusive tracer:

```
nonintrusive_logger    label    dword
                        mov      ecx, [esp+4]
                        mov      ebx, [esp]

                        cmp      dword ptr[ebx], EXCEPTION_PRIV_INSTRUCTION
                        je        __check_CL_I
                        cmp      dword ptr[ebx], EXCEPTION_GUARD_PAGE
                        jne       retkiuser

nidelta:
                        call      nidelta
                        pop       ebp
                        sub       ebp, offset nidelta

                        mov       esi, ecx

                        cmp       [esi.context_eip], end_address
                        je        remove_pguard

                        pushv     <dd    ?>
                        push      PAGE_EXECUTE_READWRITE
                        push      45bb01h
                        push      400000h
                        call      [ebp+VirtualProtect]

                        mov       ebx, [esi.context_eip]
                        cmp       ebx, bingo_address
                        jne       __donotdumpregion

                        push      bingo_size
                        push      bingo_address
                        call      dumpregiontofile

                        pushs     <"dumping address and removing PG", 13, 10>
                        call      [ebp+OutputDebugStringA]

                        push      0
                        push      esi
                        call      [ebp+NtContinue]
                        nop
                        nop
```

This is part which actually dumps region once certain conditions are met (eip == bingo\_address). If there are no our conditions yet, I dump via OutputDebugStringA instructions that caused access to my code and EIP. Then as in any other debugger, I get len of instruction and store CLI after it making sure not to lose control over nonintrusive tracer.



There are a few values used by me in loader\_type3:

```
bingo_address      equ      82e6c3h
bingo_size         equ      0082E74Fh-bingo_address
call_address       equ      0082E6B1h
ret_address        equ      call_address+5
end_address        equ      0082E750h
```

bingo\_address - address where decrypted code is (find it with loader\_bingo)  
bingo\_size - len of crypted code  
call\_address - address where call to Decrypt is  
ret\_address - adress left on stack by call (call\_address+5)  
end\_address - address of 1<sup>st</sup> valid instruction after crypted code

One example would be much better:

```
.dumped:0082E6B1      call    near ptr sub_BB07CD <-- call_address
.dumped:0082E6B6      or      eax, [eax] <-- ret_address
...
.dumped:0082E750      mov     edx, offset aSoftwareAdva_4
.dumped:0082E755      lea     ecx, [ebp-0CCh]

end_address = 82E750h
```

loader\_bingo only needs these 3 values to work and by launching loader\_bingo with those values and watching its output in DebugView we get this:

```
00000009    1.56604767 [1524] Accessing to page from : 0x00BB0B93
00000010    1.56610692 [1524] CLI reached
00000011    1.56645024 [1524] Accessing to page from : 0x00BB156B
00000012    1.56649160 [1524] CLI reached
...
00000277    3.75921321 [1524] Accessing to page from : 0x00B0A7ED
00000278    3.75996923 [1524] CLI reached
00000279    3.76135755 [1524] Accessing to page from : 0x0082E6C3
00000280    3.76140451 [1524] CLI reached
00000281    3.76171827 [1524] Accessing to page from : 0x0082E6C8
00000282    3.76175857 [1524] CLI reached
```

Well there is start of decrypted code (bingo\_address): 0x0082E6C3 if we apply these values to loader\_type3 then you will get dump of that small region:

```
seg000:00000000      mov     edx, 830084h
seg000:00000005      lea     ecx, [ebp-0C0h]
seg000:0000000B      mov     eax, 830658h
seg000:00000010      call    near ptr 0FFFF49B5h
seg000:00000015      mov     edx, [ebp-0C0h]
seg000:0000001B      mov     eax, [ebp-4]
seg000:0000001E      mov     eax, [eax+600h]
seg000:00000024      call    near ptr 0FFC35415h
seg000:00000029      mov     edx, 830084h
seg000:0000002E      lea     ecx, [ebp-0C4h]
seg000:00000034      mov     eax, 83066Ch
seg000:00000039      call    near ptr 0FFFF49B5h
seg000:0000003E      mov     edx, [ebp-0C4h]
seg000:00000044      mov     eax, [ebp-4]
seg000:00000047      mov     eax, [eax+604h]
seg000:0000004D      call    near ptr 0FFC35415h
```



```
seg000:00000052      mov     edx, 830084h
seg000:00000057      lea     ecx, [ebp-0C8h]
seg000:0000005D      mov     eax, 830680h
seg000:00000062      call   near ptr 0FFFF49B5h
seg000:00000067      mov     edx, [ebp-0C8h]
seg000:0000006D      mov     eax, [ebp-4]
seg000:00000070      mov     eax, [eax+608h]
seg000:00000076      call   near ptr 0FFC35415h
seg000:0000007B      call   near ptr 374C95h
seg000:00000080      or      eax, [eax]
```

Voila, now you have to repeat all steps for all 5 crypted code sequences. I gave you addresses so it will not be problem for you to find them. To fix dump fast I wrote one small tool called offset fixer, actually I used it to fix SVKP code\_crypt macros 5-6 months ago, but it will work like a charm here. [ofixer folder]

We take all five dumps and store them in same folder with our dump, all regions have filenames as virtual\_address.bin so offset fixer will search in current directory for \*.bin and from name of file will extract virtual\_address and apply fix to dump. All we have to do now is to patch all calls to decrypt code procedure to our decrypted code:

```
04/04/2006 09:12 PM <DIR>
04/04/2006 02:40 PM 312 82e055.bin
04/04/2006 03:20 PM 140 82e6c3.bin
04/04/2006 03:40 PM 30 00855361.bin
04/04/2006 03:56 PM 41 826c84.bin
04/04/2006 04:11 PM 61 82740c.bin
10/27/2005 12:00 AM 8,192 ofixer.EXE
04/03/2006 07:22 PM 25,030,656 test3.exe
```

Now we run ofixer.exe and select test3.exe (my final dump):



Now all we have to do is to apply patches, each call \_decrypt should be patched as shown on this picture:

0082E6AA	33D2	XOR EDX,EDX
0082E6AC	E8 0F6FE3FF	CALL test3.006655C0
0082E6B1	E9 00000000	JMP test3.0082E6C3
0082E6B6	0B	DB 0B
0082E6B7	00	DB 00
0082E6B8	00	DB 00
0082E6B9	00	DB 00
0082E6BA	00	DB 00
0082E6BB	00	DB 00



And it will jmp to decrypted code:

0082E6C3	?	BA 84008300	MOV EDX, test3.00830084
0082E6C8	.	808D 40FFFFFF	LEA ECX, DWORD PTR SS:[EBP-C0]
0082E6CE	.	B8 58068300	MOV EAX, test3.00830658
0082E6D3	.	E8 A049FFFF	CALL test3.00823078
0082E6D8	.	8B95 40FFFFFF	MOV EDX, DWORD PTR SS:[EBP-C0]
0082E6DE	.	8B45 FC	MOV EAX, DWORD PTR SS:[EBP-4]
0082E6E1	.	8B80 00060000	MOV EAX, DWORD PTR DS:[EAX+600]
0082E6E7	.	E8 EC53C3FF	CALL test3.00463A08
0082E6EC	.	BA 84008300	MOV EDX, test3.00830084
0082E6F1	.	808D 3CFFFFFF	LEA ECX, DWORD PTR SS:[EBP-C4]
0082E6F7	.	B8 6C068300	MOV EAX, test3.0083066C
0082E6FC	.	E8 7749FFFF	CALL test3.00823078
0082E701	.	8B95 3CFFFFFF	MOV EDX, DWORD PTR SS:[EBP-C4]
0082E707	.	8B45 FC	MOV EAX, DWORD PTR SS:[EBP-4]
0082E70A	.	8B80 04060000	MOV EAX, DWORD PTR DS:[EAX+604]
0082E710	.	E8 C353C3FF	CALL test3.00463A08
0082E715	.	BA 84008300	MOV EDX, test3.00830084
0082E71A	.	808D 38FFFFFF	LEA ECX, DWORD PTR SS:[EBP-C8]
0082E720	.	B8 80068300	MOV EAX, test3.00830680
0082E725	.	E8 4E49FFFF	CALL test3.00823078
0082E72A	.	8B95 38FFFFFF	MOV EDX, DWORD PTR SS:[EBP-C8]
0082E730	.	8B45 FC	MOV EAX, DWORD PTR SS:[EBP-4]
0082E733	.	8B80 08060000	MOV EAX, DWORD PTR DS:[EAX+608]
0082E739	.	E8 9A53C3FF	CALL test3.00463A08
0082E73E	.v	E9 0D000000	JMP test3.0082E750
0082E743	.	0B	DB 0B
0082E744	.	00000001	DD test3.01000000
0082E748	.	0B	DB 0B
0082E749	.	00008300	DD test3.00830000

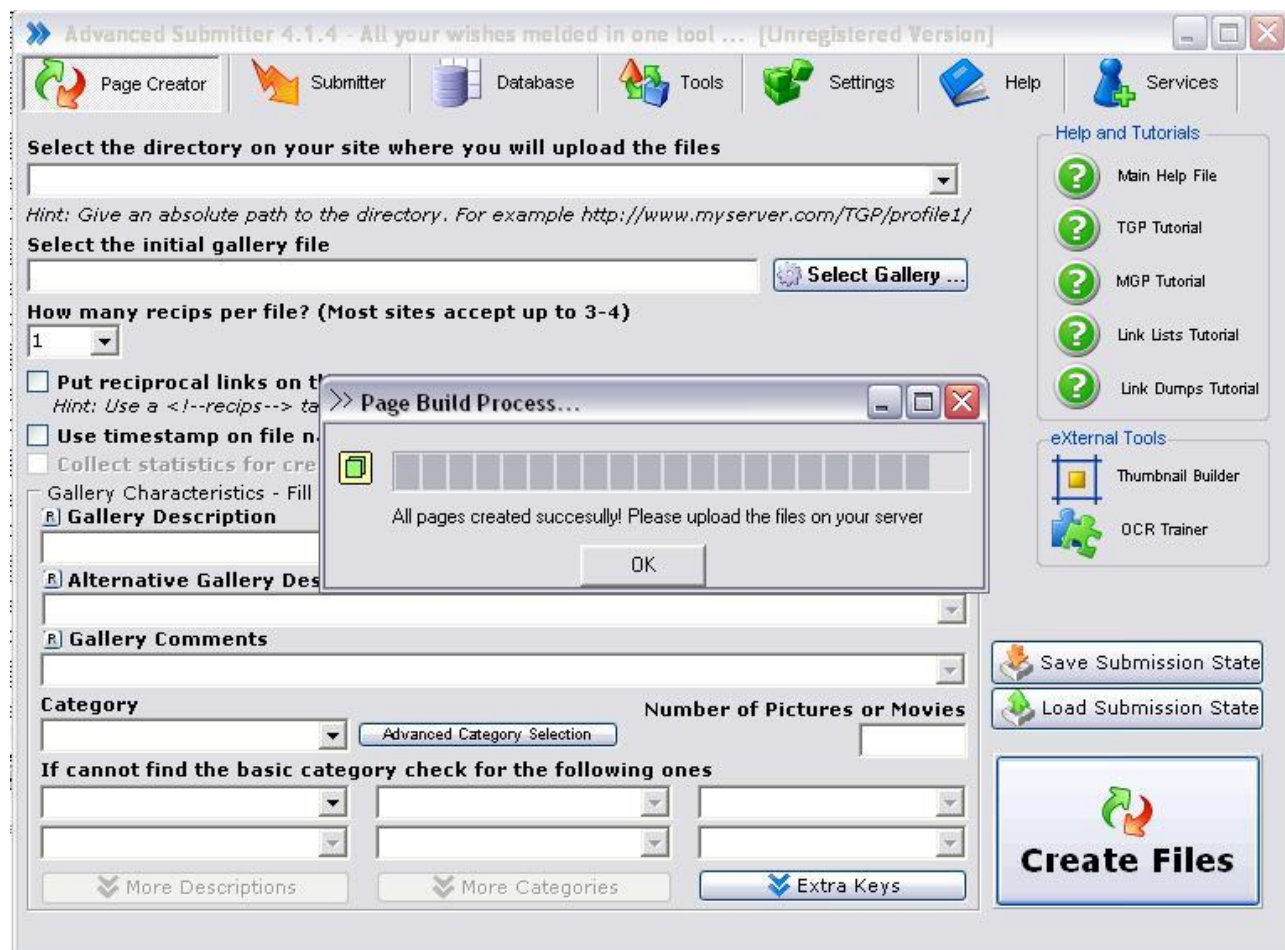
In decrypted code we fix call \_EncryptCode to jmp \_\_next\_valid\_code:

0082E750	?	BA 84008300	MOV EDX, test3.00830084
0082E755	.	808D 34FFFFFF	LEA ECX, DWORD PTR SS:[EBP-CC]
0082E75B	.	B8 94068300	MOV EAX, test3.00830694
0082E760	.	E8 1349FFFF	CALL test3.00823078
0082E765	.	8B85 34FFFFFF	MOV EAX, DWORD PTR SS:[EBP-CC]
0082E76B	.	BA A4068300	MOV EDX, test3.008306A4
0082E770	.	E8 8F66BDFD	CALL test3.00404E04

You have to repeat these steps for all 5 code blocks, you have all addresses, you have all dumped regions all you have to do is to apply them using [ofixer.exe].



And we are done, we run unpacked application and try to produce some “gallery” and we see that we have unpacked it perfectly:



That’s it... if you want to crack it go ahead; I was interested in themida ring0 protection, not in application itself.



## 5. Conclusion

Very good protection, very good, if I was software developer I would use themida 1.0.0.5 to protect my application. That would for sure nook 90% of reversing community and your application would remain uncracked for a long time J It was fun to play with themida, just to show myself that I can do it. Now I will focus on some other protector J

**S verom u Boga, deroko/ARTeam**

**All the code provided with this tutorial is free for public use, just make a greetz to the authors and the ARTeam if you find it useful to use. Don't use these concepts for making illegal operation, all the info here reported are only meant for studying and to help having a better knowledge of application code security techniques.**



## 6. References/Tools

- [1] Unpacking Armadilloed DLL, deroko, <http://tutorials.accessroot.com>
- [2] DebugView, Mark Russinovich, [www.sysinternals.com](http://www.sysinternals.com)
- [3] AdvancedSubmitter 4.1.4, Target, <http://root.accessroot.com/tools/AdvSubmitter-v4.1.4-Unregistered.exe>
- [4] Unpacking and Dumping ExeCryptor and coding loader, deroko, <http://tutorials.accessroot.com>
- [5] Anti-Anti-Dump and nonintrusive tracers, deroko, <http://tutorials.accessroot.com>
- [6] Microsoft © Windows ® Internals, Mark Russinovich, David Solomon, Microsoft Press



## 7. Greetings

I wish to tank all the ARTeam members for sharing their knowledge, to 29a virus writing group for one of the best e-zines, to my friends from phearless e-zine, and to all great coders out there... and of course you for reading this article.



<http://cracking.accessroot.com>